

A Complete Refinement Procedure for Regular Separability of Context-Free Languages

Graeme Gange^a, Jorge A. Navas^b, Peter Schachte^a, Harald Søndergaard^a, Peter J. Stuckey^a

^a *Department of Computing and Information Systems, The University of Melbourne, Vic. 3010, Australia*

^b *NASA Ames Research Center, Moffett Field, CA 94035, USA*

Abstract

Often, when analyzing the behaviour of systems modelled as context-free languages, we wish to know if two languages overlap. To this end, we present an effective semi-decision procedure for regular separability of context-free languages, based on counter-example guided abstraction refinement. We propose two refinement methods, one inexpensive but incomplete, and the other complete but more expensive. We provide an experimental evaluation of this procedure, and demonstrate its practicality on a range of verification and language-theoretic instances.

Keywords: abstraction refinement, context-free languages, regular approximation, separability

1. Introduction

We address the problem of checking whether two given context-free languages L_1 and L_2 are disjoint. This is a fundamental language-theoretical problem. It is of interest in many practical tasks that call for some kind of automated reasoning about programs. This can be because program behaviour is modelled using context-free languages, as in software verification approaches that try to capture a program's control flow as a (pushdown-system) path language. Or it can be because we wish to reason about string-manipulating programs, as is the case in software vulnerability detection problems, where various types of injection attack have to be modelled.

The problem of context-free disjointness is of course undecidable, but semi-decision procedures exist for non-disjointness. For example, one can systematically generate strings w over the intersection $\Sigma_1 \cap \Sigma_2$, where Σ_1 is the alphabet of L_1 and Σ_2 is that of L_2 . If some w belongs to both L_1 and L_2 , answer “yes, the languages overlap.” It follows that no semi-decision procedure exists for disjointness. However, semi-decision procedures exist for the stronger requirement of being separable by a regular language. For example, one can systematically generate (representations of) regular languages over $\Sigma_1 \cup \Sigma_2$, and, if some such language R is found to satisfy $L_1 \subseteq R \wedge L_2 \subseteq \overline{R}$, answer “yes, the languages are disjoint”.

A radically different approach, which we will follow here, uses so-called *counter-example guided abstraction refinement* (CEGAR) [5] of regular over-approximations. The scheme is based on repeated approximation refinement, like so:

1. *Abstraction:* Compute *regular approximations* R_1 and R_2 such that $L_1 \subseteq R_1$ and $L_2 \subseteq R_2$. (Here R_1 and R_2 are regular languages, represented using regular expressions, say.)
2. *Verification:* Check whether the intersection of R_1 and R_2 is empty using a decision procedure for regular expressions. If $R_1 \cap R_2 = \emptyset$ then $L_1 \cap L_2 = \emptyset$, so answer “the languages are disjoint.” If $w \in (R_1 \cap R_2)$, $w \in L_1$, and $w \in L_2$ then $L_1 \cap L_2 \neq \emptyset$, so answer “the languages overlap” and provide w as a witness. Otherwise, go to step 3.

Email addresses: gkgange@unimelb.edu.au (Graeme Gange), jorge.a.navaslaserna@nasa.gov (Jorge A. Navas), schachte@unimelb.edu.au (Peter Schachte), harald@unimelb.edu.au (Harald Søndergaard), pstuckey@unimelb.edu.au (Peter J. Stuckey)

3. *Refinement*: Produce new regular approximations R'_1 and R'_2 such that for each R'_i , $i \in \{1, 2\}$, we have $L_i \subseteq R'_i \subseteq R_i$, and $R'_i \subset R_i$ for some i . Update the approximations $R_1 \leftarrow R'_1$, $R_2 \leftarrow R'_2$, and go to step 2.

For the abstraction step, note that regular approximations exist, trivially. For the verification step, we could also take advantage of the fact that the class of context-free languages is closed under intersection with regular languages; however, this does not eliminate the need for a refinement procedure. For the refinement step, note that there is no indication of *how* the tightening of approximations should be done; indeed that is the focus of this paper. The step is clearly well-defined since, if $L \subset R$, there is always a regular language $R' \subset R$ such that $L \subseteq R'$.

For a given language L there may well be an infinite chain $R_1 \supset R_2 \supset \dots \supset L$ of regular approximations. This is a source of possible non-termination of the CEGAR scheme. An interesting question therefore is: Are there refinement techniques that can guarantee termination at least when L_1 and L_2 are *regularly separable* context-free languages, that is, when there exists a regular language R such that $L_1 \subseteq R$ and $L_2 \subseteq \bar{R}$?

In this paper we answer this question in the affirmative. We propose a refinement procedure which can ensure termination of the CEGAR-based loop assuming the context-free languages involved are regularly separable. In this sense we provide a refinement procedure which is *complete* for regularly separable context-free languages. Of course the question of regular separability of context-free languages is itself undecidable [18]. The method that we propose can also be used on language instances that are not regularly separable, and it will often decide such instances successfully. However, in this case, it does not come with a termination guarantee.

Contribution. The paper rests on regular approximation ideas by Nederhof [16] and we utilise the efficient *pre** algorithm [7] for intersecting (the language of) a context-free grammar with (that of) a finite-state automaton. We propose a novel refinement procedure for a CEGAR inspired method to determine whether context-free languages are disjoint, and we prove the procedure complete for determining regular separability. In the context of regular approximation, where languages must be over-approximated using *regular* languages, separability is equivalent to regular separability, so the completeness means that the refinement procedure is optimal. On the practical side, the method has important applications in software verification and security analysis. We demonstrate its feasibility through an experimental evaluation.

Outline. Section 2 introduces concepts, notation and terminology used in the paper. It also recapitulates relevant results about regular separability and language representations. Section 3 proposes a new procedure for regular approximation of context-free languages and shows that the procedure is complete, in the sense that it proves the separability for any pair of regularly separable context-free languages. Section 4 provides an example. In Section 5 we place our method in context, comparing with previously proposed refinement techniques. In Section 6 we evaluate the method empirically, comparing an implementation with the most closely related tool. Section 7 concludes. The appendices contain more peripheral implementation detail and a description of test cases used for the experimental evaluation.

2. Preliminaries

In this section we recall the notion of regular separability and introduce a concept of “star-contraction” for regular expressions.

2.1. Regular and Context-Free Languages

We first recall some basic formal-language concepts. These are assumed to be well understood—the only purpose here is to fix our terminology and notation. Given an alphabet Σ , Σ^* denotes the set of all finite strings of symbols from Σ . The string y is a *substring* of string w iff $w = xyz$ for some (possibly empty) strings x and z .

The *regular expressions* over an alphabet $\Sigma = \{a_1, \dots, a_n\}$ are \emptyset , ε , a_1, \dots, a_n , together with expressions of form $e_1|e_2$, $e_1 \cdot e_2$, and e^* , where e , e_1 and e_2 are regular expressions. Here $|$ denotes union, \cdot denotes

language concatenation, and $*$ is Kleene star. As is common, we will often omit \cdot , so that juxtaposition of e_1 and e_2 denotes concatenation of the corresponding languages. Given a finite set $E = \{e_1, \dots, e_k\}$ of regular expressions, we let $\|E$ stand for the regular expression $e_1 | \dots | e_k$ (in particular, $\|\emptyset = \emptyset$). We let Reg_Σ denote the set of regular expressions over alphabet Σ .

A (non-deterministic) *finite-state automaton* is a quintuple $\langle Q, \Sigma, \delta, q_0, F \rangle$ where Q is the set of states, Σ is the alphabet, δ is the transition relation, q_0 is the start state, and F is the set of accept states. The presence of (q, x, q') in $\delta \subseteq Q \times \Sigma \times Q$ indicates that, on reading symbol x while in state q , the automaton may proceed to state q' . If δ is a total function, that is, if for all $q \in Q, x \in \Sigma, |\{q' \mid (q, x, q') \in \delta\}| = 1$, then the automaton is *deterministic*.

A language which can be expressed as a regular expression (or equivalently, has a finite-state automaton that recognises it) is *regular*. The language recognised by automaton A is written as $\mathcal{L}(A)$. Similarly, $\mathcal{L}(e)$ is the language denoted by regular expression e .

A *context-free grammar*, or CFG, is a quadruple $G = \langle V, \Sigma, P, S \rangle$, where V is the set of variables (non-terminals), S is the start symbol, and P is the set of productions (or rules). Each production is of form $X \rightarrow \alpha$ with $X \in V$ and $\alpha \in (V \cup \Sigma)^*$. If $X \rightarrow \alpha$ is a production in P then, for all $\beta, \gamma \in (V \cup \Sigma)^*$, we say that $\beta X \gamma$ *yields* $\beta \alpha \gamma$, written $\beta X \gamma \Rightarrow \beta \alpha \gamma$. The language *generated* by G is $\mathcal{L}(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$, where \Rightarrow^* is the reflexive transitive closure of \Rightarrow . A set of strings is a context-free language (CFL) iff it is generated by some CFG.

In algorithms we represent regular languages using finite-state automata, and CFLs using CFGs. When there is little risk of confusion, we ignore the distinction between a language and its representation. Hence we may, for example, apply set operations to (the transition relation of) a finite-state automaton.

2.2. Regular Separability

As our approach uses regular approximations to context-free languages, we cannot hope to prove separation for arbitrary disjoint pairs of context-free languages. Instead we focus on pairs of *regularly separable* languages.

Definition 1 (Regularly separable). Two context-free languages L_1 and L_2 are *regularly separable* iff there exists a regular language R such that $L_1 \subseteq R$ and $L_2 \subseteq \overline{R}$ where \overline{R} is the complement of R .

It will be useful to have a slightly different view of separability:

Definition 2 (Separating pair). Given a pair (L_1, L_2) of context-free languages, a pair (R_1, R_2) of regular languages form a *separating pair* for (L_1, L_2) iff $L_1 \subseteq R_1$, $L_2 \subseteq R_2$, and $R_1 \cap R_2 = \emptyset$.

Lemma 1. *Context-free languages L_1 and L_2 are regularly separable iff there exists some separating pair (R_1, R_2) for (L_1, L_2) .*

Proof. If (R_1, R_2) is a separating pair then $L_1 \subseteq R_1$, and $L_2 \subseteq R_2 \subseteq \overline{R_1}$, so L_1 and L_2 are regularly separable. Conversely, if the regular language R separates L_1 and L_2 then we have $L_1 \subseteq R$, $L_2 \subseteq \overline{R}$, and $R \cap \overline{R} = \emptyset$. So (R, \overline{R}) is a separating pair. \square

To see that there are disjoint context-free languages that are not regularly separable, consider $L = \{a^n b^n \mid n \geq 0\}$. Both L and \overline{L} are non-regular context-free languages, and therefore not regularly separable. The problem of checking whether a pair of context-free languages is regularly separable is undecidable [11].

2.3. Star-Contraction

Definition 3 (Union-free regular language). A regular expression is *union-free* iff it does not use the union operation. A regular language is *union-free regular* if it can be written as a union-free regular expression. We use Reg'_Σ to denote the set of union-free regular expressions.

Union-free regular languages are also known as *star-dot regular* languages [3].

Definition 4 (Union-free decomposition). A union-free decomposition [14] of a regular language R is a finite set of union-free regular languages R_1, \dots, R_n such that $R = R_1 \cup \dots \cup R_n$.

Theorem 1 (Nagy [14]). *Every regular language R admits some finite union-free decomposition.*

Theorem 1 is not surprising; it utilises the well-known equivalence $(r_1|r_2)^* = (r_1^*r_2^*)^*$.

The following concept is central to this paper’s ideas. For a given union-free language, it is convenient to consider particular sets of sub-languages:

Definition 5 (Star-contraction). The star-contraction $\kappa(P)$ of a union-free regular expression is the set of languages obtained by replacing some subset of $*$ -enclosed subterms in P with ε . The naive construction is confounded by the presence of nested $*$ operators, as distinct portions of the subterms may occur in each outer repetition. the star-contraction is defined as:

$$\begin{aligned} \kappa(\mathbf{a}) &= \{\mathbf{a}\} && \text{for } \mathbf{a} \in \Sigma \cup \{\varepsilon\} \\ \kappa(e^*) &= \{(\|E)^* \mid E \subseteq \kappa(e)\} \\ \kappa(e_1 \cdot e_2) &= \{r_1 \cdot r_2 \mid r_1 \in \kappa(e_1) \wedge r_2 \in \kappa(e_2)\} \end{aligned}$$

Example 1. Consider the union-free regular expression $e = (\mathbf{ab}^*\mathbf{c}^*)^*$. The star-contraction of the parenthesised term is $\kappa(\mathbf{ab}^*\mathbf{c}^*) = \{\mathbf{a}, \mathbf{ab}^*, \mathbf{ac}^*, \mathbf{ab}^*\mathbf{c}^*\}$.

The star-contraction of e (after elimination of equivalent languages) is then:

$$\kappa(e) = \{\varepsilon, \mathbf{a}^*, (\mathbf{ab}^*)^*, (\mathbf{ac}^*)^*, (\mathbf{ab}^*|\mathbf{ac}^*)^*, (\mathbf{ab}^*\mathbf{c}^*)^*\}.$$

Note how the elements $r \in \kappa(e)$ with $r \neq e$ make particular subsets of $\mathcal{L}(e)$ explicit. For example, \mathbf{a}^* is the subset that makes no use of \mathbf{b} or \mathbf{c} , whereas $(\mathbf{ab}^*|\mathbf{ac}^*)^*$ is the set of words in which \mathbf{b} and \mathbf{c} are not adjacent.

We later use $\kappa(R)$, that is, κ applied to a regular language, to denote $\kappa(e_1) \cup \dots \cup \kappa(e_m)$, where $\{e_1, \dots, e_m\}$ is some (arbitrary) union-free decomposition of R . Note that the star-decomposition of a regular language is not unique; different union-free decompositions give rise to different star-contractions. We assume, for a regular language R , $\kappa(R)$ deterministically returns *some* valid star-contraction of R .

$\kappa(R)$ has several properties which will be useful in the following:

Proposition 1. *For any regular language R :*

1. $\kappa(R)$ is finite.
2. For every regular expression $e \in \kappa(R)$, $\mathcal{L}(e) \subseteq R$.
3. $\bigcup\{\mathcal{L}(e) \mid e \in \kappa(R)\} = R$

3. Refining Regular Abstractions

We now describe the main idea behind the refinement phase. We are interested in the intersection of a finite set of languages, but without loss of generality, we consider the intersection of just two context-free languages L_1 and L_2 (provided as context-free grammars).

We assume a decision procedure that returns “no” if $\mathcal{L}(A_1) \cap \mathcal{L}(A_2) = \emptyset$ or returns a witness w if $w \in \mathcal{L}(A_1) \cap \mathcal{L}(A_2) \neq \emptyset$, where A_1 and A_2 are finite-state automata recognising regular languages R_1 and R_2 , respectively (that is, $\mathcal{L}(A_1) = R_1$ and $\mathcal{L}(A_2) = R_2$). Moreover, our refinement procedure will require the solving of constraints of the form $A = A_1 \setminus A_2$ where A , A_1 and A_2 are finite-state automata, that is, A recognises $\mathcal{L}(A_1) \cap \overline{\mathcal{L}(A_2)}$.

Assume that at some point we have regular approximations $R_1 \supseteq L_1$ and $R_2 \supseteq L_2$, and we have found some witness w such that $w \in R_1 \cap R_2$, but $w \notin L_1 \cap L_2$. There are three cases to consider:

$$(1) \ w \notin L_1 \wedge w \in L_2 \quad (2) \ w \in L_1 \wedge w \notin L_2 \quad (3) \ w \notin L_1 \wedge w \notin L_2$$

For cases (1) and (2) we should refine R_1 and R_2 , respectively. For case (3) we could choose to refine either R_1 or R_2 , or both. In our implementation, we always refine all the regular approximations.

If $w \notin L_i$ then a straightforward refinement is to produce a new abstraction $R_i \setminus \{w\}$ in place of R_i . However, this refinement process will rarely converge, as we can exclude only finitely many strings in finite time. We must instead formulate a refinement procedure which generalizes a counterexample to an infinite set of words.

3.1. Star-generalizations

The refinement procedure in this section operates by taking the regular expression recognizing the single counterexample w , and progressively augmenting it with $*$ operators while ensuring the counterexample and query remain disjoint.

Definition 6 (Star-generalization). The star-generalizations of a word w is the set Ξ of regular expressions given by

$$\begin{aligned} \Xi(\varepsilon) &= \{\varepsilon\} \\ \Xi(x) &= \{x, x^*\} \text{ for } x \in \Sigma \\ \Xi(\alpha_1 \dots \alpha_n) &= \{\alpha_1 \dots \alpha_n, (\alpha_1 \dots \alpha_n)^*\} \cup \left\{ e_1 e_2, (e_1 e_2)^* \mid \begin{array}{l} e_1 \in \Xi(\alpha_1 \dots \alpha_i), \\ e_2 \in \Xi(\alpha_{i+1} \dots \alpha_n), \\ i \in [1, n-1] \end{array} \right\} \end{aligned}$$

A star-generalization of w is a language which can be constructed by adding (nested) unbounded repetition of intervals in w .

We shall represent a star-generalization as a pair $\langle w, S \rangle$ of a word $w = w_1 \dots w_n$ and the set S of ranges covered by $*$ -operators. We denote such a range by the pair (i, j) of the first and last index covered by the operator. S must satisfy the constraints

$$\begin{aligned} \forall (i, j) \in S . i, j \in [0, n], i < j \\ \forall (i, j), (i', j') \in S . j \leq i' \vee j' \leq j \end{aligned} \tag{1}$$

This ensures the set of $*$ -enclosed ranges are well-formed. We shall use $\mathcal{L}(\langle w, S \rangle)$ to denote the language that results from such a generalization.

Definition 7 (Star-generalization with respect to a language). The star-generalizations of w with respect to some language L (denoted $\Xi_L(w)$) is the set of star-generalizations of w which are contained in L . Formally, this can be expressed as

$$\Xi_L(w) = \{r \mid r \in \Xi(w), \mathcal{L}(r) \subseteq L\}.$$

Definition 8 (Maximal star-generalization). A maximal star-generalization of w with respect to some language L is a star-generalization r of w such that there is no other star-generalization r' with $\mathcal{L}(r) \subset \mathcal{L}(r') \subseteq L$.

A maximal star-generalization is not necessarily unique. Consider generalizing ab with respect to $(a^*b|ab^*)$ – both a^*b and ab^* are incomparable maximal generalizations.

A greedy procedure for constructing a star-generalization is given in Figure 1. The algorithm takes as input a witness $w \in R_1 \cap R_2$, context-free language L such that $w \notin L$, and L 's regular approximation A (either R_1 or R_2). The procedure begins with a trivial star-generalization recognizing only w . P stores the set of (i, j) pairs where a $*$ -operation may be introduced without causing the generalization to be malformed. At each step, we add one of the candidate operations to the generalization, then remove any pairs from P which are no longer feasible (because they violate the nestedness requirement). Following (1), this is the set of pairs (i', j') such that $i < i' < j < j' \vee i' < i < j' < j$.

It is worth pointing out that this refinement procedure is *anytime*: If for some reason it would seem necessary or advantageous, one can, without compromising correctness, interrupt the while loop having considered only a subset of the possible $*$ -augmentations, thereby settling for a smaller generalization.

Example 2. Let $L = \{a^i b^{i+1} \mid i \geq 0\}$ be (currently) approximated by a^*b^* , and let the witness $w \notin L$ be aab . To refine the regular approximation, $\text{refine}(w, L, a^*b^*)$ begins with the trivial star-generalization $\langle w, \emptyset \rangle$. We greedily augment the counterexample with $*$ -operations, in this case following lexicographic order, with the accumulated language here described with a regular expression:

```

refine( $w, L, A$ )
1:  let  $w$  be  $x_1 \cdot x_2 \cdots x_n$ 
2:   $S := \emptyset$ 
3:   $P := \{(i, j) \mid i, j \in [0, n], i < j\}$ 
4:  while  $P \neq \emptyset$ 
5:    choose  $(i, j) \in P$ 
6:     $S' := S \cup \{(i, j)\}$ 
7:     $P := P \setminus \{(i, j)\}$ 
8:    if  $(\mathcal{L}(\langle w, S' \rangle) \cap L = \emptyset)$ 
       $S := S'$ 
       $P := \left\{ (i', j') \mid \begin{array}{l} (i', j') \in P \\ \wedge (j \leq i' \vee j' \leq j) \\ \wedge (j' \leq i \vee j \leq j') \end{array} \right\}$ 
9:  return  $A \setminus \mathcal{L}(\langle w, S \rangle)$ 

```

Figure 1: Refining a regular approximation greedily by removing a maximal star-generalization of some counterexample w .

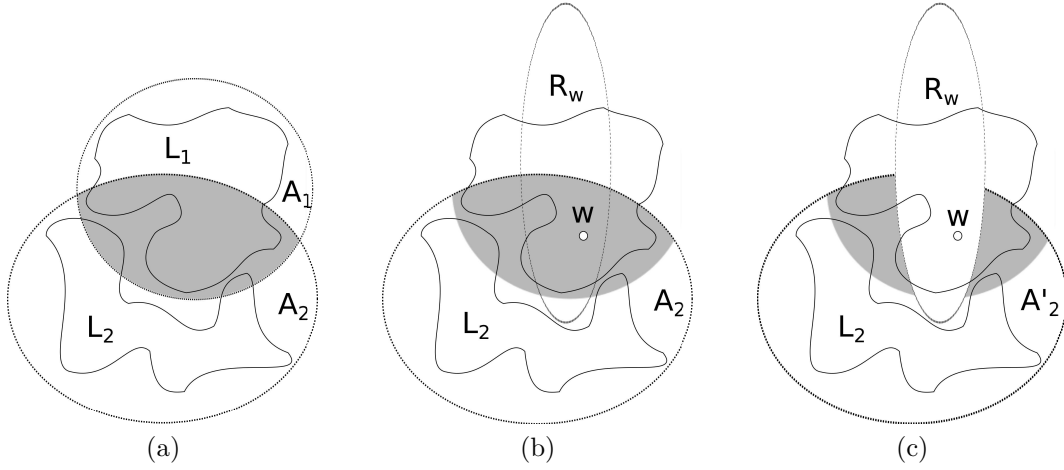


Figure 2: (a) A pair of context free languages L_1, L_2 and their initial approximations A_1, A_2 . The intersection of the approximations is shaded. (b) A counterexample $w \in A_1 \cap A_2$ has been identified. As $w \notin L_2$, we build a generalization R_w such that $\{w\} \subseteq R_w \subseteq \overline{L_2}$. (c) L_2 's new approximation is $A'_2 = A_2 \setminus R_w$.

*-augmentation	$\mathcal{L}(\langle w, S \rangle)$
	aab
(0, 1)	include, obtaining a*ab
(1, 2)	exclude, as b $\in L$
(2, 3)	exclude, as abb $\in L$
(0, 2)	exclude, as b $\in L$
(1, 3)	include, obtaining a*(ab)*
(0, 3)	include, obtaining (a*(ab)*)*

The complement of the resulting language is $(a^*ab)^*b(a|b)^*$. The language returned by $\text{refine}(w, L, a^*b^*)$ is (represented by) the intersection automaton of this and the initial approximation a^*b^* , yielding $bb^*|aa^*bbb^*$. This is the new, improved, regular approximation of L . By construction, it does not contain **aab**, but more importantly, along with **ab**, infinitely many other strings have been discarded from the previous approximation a^*b^* , namely all those strings containing at least one **a** and exactly one **b**.

The overall flow of the refinement step is illustrated in Figure 2. Figure 1's refinement procedure has these properties:

1. it is *sound*: $L_i \subseteq (R_i \setminus \mathcal{L}(R)) \subseteq R_i$.
2. it *terminates*: the while loop will be executed at most $\frac{n(n+1)}{2}$ times.
3. it is *progressive*: the same witness w cannot be produced again upon successive calls to **refine**.

The refinement algorithm involves a check (line 8) to see if a regular and a context-free language overlap. This problem is decidable in polynomial time¹.

The following theorem is critical to the completeness result, showing the relationship between κ and Ξ . Note that here we are intersecting sets of languages, rather than the languages themselves.

Lemma 2. *Let e be a union-free regular expression, and κ and Ξ be the star-contraction and star-generalizations given in Definitions 5 and 6. Then for all $w \in \mathcal{L}(e)$, $\kappa(e) \cap \Xi(w) \neq \emptyset$.*

Proof. Assume $e = \{a\}$, $a \in \Sigma \cup \{\varepsilon\}$. Then $w = a$. From the definitions, we have $\{a\} \in \kappa(e)$, and $\{a\} \in \Xi(w)$.

Assume $e = e_1 \cdot e_2$, such that the induction hypothesis holds on e_1 and e_2 . Consider some word $w \in \mathcal{L}(e)$. We can partition w into $w_1 \cdot w_2$, such that $w_1 \in \mathcal{L}(e_1)$, $w_2 \in \mathcal{L}(e_2)$. By the induction hypothesis, there is some r_1, r_2 such that $r_1 \in \kappa(e_1) \cap \Xi(w_1)$, $r_2 \in \kappa(e_2) \cap \Xi(w_2)$. As $r_1 \in \kappa(e_1)$ and $r_2 \in \kappa(e_2)$, from Definition 5 we have $r_1 \cdot r_2 \in \kappa(e)$. By Definition 6, we also have $r_1 \cdot r_2 \in \Xi(w)$. Therefore $r_1 \cdot r_2 \in \kappa(e) \cap \Xi(w)$.

Assume $e = e'^*$, for some e' satisfying the induction hypothesis. Consider some word $w \in e$. We can partition w into $w_1 \dots w_k$, such that each $w_i \in e'$. By the induction hypothesis, each w_i admits some star-generalization $r_i \in \kappa(e') \cap \Xi(w_i)$. Consider the generalization r given by

$$r = (r(w_1)^* \dots r(w_k)^*)^*.$$

By Definition 6, $r \in \Xi(w)$. r is equivalent to $(r(w_1) \cup \dots \cup r(w_k))^*$, which is in $\kappa(e)$. Therefore, $r \in \Xi(w) \cap \kappa(e)$. \square

Theorem 2. *Let R be a regular language, and κ and Ξ be the star-contraction and star-generalizations given in Definitions 5 and 6. Then for all $w \in R$, $\kappa(R) \cap \Xi(w) \neq \emptyset$.*

Proof. As noted in Section 2.3, the star-contraction of a regular language R is computed based on some union-free decomposition $E = \{e_1, \dots, e_n\}$ of R . Consider $w \in R$. Therefore, there is some $e \in E$ such that $w \in \mathcal{L}(e)$. By Lemma 2, $\kappa(e) \cap \Xi(w) \neq \emptyset$. As $\kappa(e) \subseteq \kappa(R)$, we have $\kappa(R) \cap \Xi(w) \neq \emptyset$. \square

3.2. Epsilon-generalization

The notion of star-generalization, while useful for reasoning, does not integrate well into existing automaton algorithms. In this section, we introduce a slightly different form of generalization.

Let $\nu(w)$ denote the automaton recognizing the single word $w = x_1 \dots x_n$. We have $\nu(x_1 \dots x_n) = \langle Q, \Sigma, \delta, q_0, \{q_n\} \rangle$, with $Q = \{q_0, \dots, q_n\}$ and $\delta = \{(q_{i-1}, x_i, q_i) \mid i \in [1, n]\}$.

Definition 9 (Epsilon-generalization). *An epsilon-generalization of w is any language obtained by augmenting the transition function of $\nu(w)$ with additional edges $E \cup P$, given by:*

$$E \subseteq \{(q_i, \varepsilon, q_j) \mid i < j\}$$

$$P \subseteq \{(q_{j-1}, w_j, q_i) \mid i < j\}$$

¹The algorithm *pre** described in [6, 7] has a time complexity of $O(|R| \times |Q|^3)$ and space complexity of $O(|R| \times |Q|^2)$, where $|R|$ is the number of productions in the context-free grammar and $|Q|$ is the number of states in the automaton. We use this algorithm in our implementation.

That is, we may only introduce epsilon-transitions forwards; backwards transitions always consume the same input character as the original outgoing transition from the source state. We shall use $\text{Gen}(w)$ to denote the set of epsilon-generalizations of w . Note that where $\Xi(w)$ is a set of languages, $\text{Gen}(w)$ is a set of automata. Similarly, $\text{Gen}_L(w)$ denotes the set of epsilon-generalizations with respect to some language L .

Lemma 3. *Let $A_1 = \langle Q_1, \Sigma, \delta_1, q_{10}, \{q_{1n}\} \rangle$ and $A_2 = \langle Q_2, \Sigma, \delta_2, q_{20}, \{q_{2n}\} \rangle$ be (nondeterministic) finite-state automata such that q_{1n} has no outgoing edges.*

Then the automaton $A_1 \circ A_2 = \langle (Q_1 \cup Q_2) \setminus \{q_{1n}\}, \Sigma, \delta_1 \cup \delta_2[q_{1n} \mapsto q_{20}], \{q_{2n}\} \rangle$ recognizes the language $\mathcal{L}(A_1) \cdot \mathcal{L}(A_2)$.

Proof. Let $A = A_1 \circ A_2$. Assume $w \in \mathcal{L}(A_1) \cdot \mathcal{L}(A_2)$. Then $w = w_1 \cdot w_2$, for some $w_1 \in \mathcal{L}(A_1), w_2 \in \mathcal{L}(A_2)$. So there is some path from q_{10} to q_{20} matching w_1 in $A_1 \circ A_2$, and a path from q_{20} to q_{2n} matching w_2 . Therefore w is recognized by A .

Assume w is recognized by A . There are no transitions from states in Q_1 to states in Q_2 except q_{20} ; therefore, any path from q_{10} to q_{2n} in A must pass through q_{20} . There are no transitions from states in Q_2 to states in Q_1 (as q_{1n} had no outgoing edges). Therefore, once reaching a state in Q_2 , a path through A must remain in Q_2 .

Hence we can divide the path through A into a prefix, following transitions exclusively in A_1 and reaching q_{20} , and a suffix from q_{20} to q_{2n} following transitions exclusively in A_2 . Therefore, $w \in \mathcal{L}(A_1) \cdot \mathcal{L}(A_2)$. \square

Theorem 3. *For any word w , $e \in \Xi(w)$, there is some $A \in \text{Gen}(w)$ such that $\mathcal{L}(e) = \mathcal{L}(A)$. That is, star-generalization of w may be expressed by an equivalent epsilon-generalization.*

Proof. Consider $e \in \Xi(w)$.

Assume $e = \mathbf{a}$, $\mathbf{a} \in \Sigma \cup \{\varepsilon\}$. $\nu(\mathbf{a}) \in \text{Gen}(w)$, and $\mathcal{L}(\nu(\mathbf{a})) = \mathcal{L}(e)$. Also, the final state of $\nu(\mathbf{a})$ has no outgoing transitions.

Assume that for all expressions e' of up to depth k , if $e' \in \Xi(w)$ there is some $A \in \text{Gen}(w)$ such that $\mathcal{L}(A) = \mathcal{L}(e')$, and the accept state of A has no outgoing transitions. Consider some star-generalization $e \in \Xi(w)$ of depth $k + 1$.

Assume $e = e'^*$ for some generalization $e' \in \Xi(w)$. As e has depth $k + 1$, e' is of depth k . Then there is some automaton $A = \langle Q, \Sigma, \delta, q_0, \{q_n\} \rangle \in \text{Gen}(w)$ such that $\mathcal{L}(A) = \mathcal{L}(e')$. We construct a new automaton A' with transition relation δ'

$$\delta' = \delta \cup \{q_0 \xrightarrow{\varepsilon} q_n\} \cup \{(q_{j-1} \xrightarrow{w_j} q_0) \mid (q_j \xrightarrow{\varepsilon^*} q_n) \in \delta\}$$

The added transitions are of the form permitted by Definition 9. As A is an epsilon-generalization of w , A' is also a valid epsilon-generalization. As the accept state of A had no outgoing transitions, and we have not added any transitions beginning at q_n , the accept state of A' also has no outgoing transitions. We now consider the language recognized by A' . Assume some word w is in $\mathcal{L}(e'^*)$. Then either $w = \varepsilon$, or $w = w_1 \dots w_m$ such that $w_i \in \mathcal{L}(e') \setminus \{\varepsilon\}$. If $w = \varepsilon$, then w is recognized by A' (by $q_0 \xrightarrow{\varepsilon} q_n$). Otherwise, each w_i is recognized by some path p_i from q_j to q_k in A , such that q_n is reachable from q_k by ε -transitions. Let $q_{k'}$ be the second-last state in p_i . By construction, there must be some alternate transition from $q_{k'}$ to q_0 in A' ; then there must be some path following w_i from q_0 to q_0 in A' . Therefore, w is recognized by A' . Now assume there is some word $w \neq \varepsilon$ recognized by A' . We can partition w into sub-words $w_1 \dots w_m$ such that the path of each w_i with $i < m$ starts at q_0 , makes its final transition via an introduced edge, and uses no other introduced edges. As the path corresponding to w_i finishes with an introduced edge, there must be some corresponding path from q_0 to q_n in A . So $w_i \in \mathcal{L}(e')$ for $i < m$. And as the path corresponding to w_m starts at q_0 and does not use any introduced edges, $w_m \in \mathcal{L}(e')$. Therefore, $w \in \mathcal{L}(e'^*) = \mathcal{L}(e)$. Therefore $\mathcal{L}(A') = \mathcal{L}(e'^*) = \mathcal{L}(e)$.

Assume $e = e_1 \cdot e_2$ for $e_1 \in \Xi(w_1)$, $e_2 \in \Xi(w_2)$ and $w = w_1 w_2$. As e_1 and e_2 have depth at most k , there exists $A_1 \in \text{Gen}(w_1)$ and $A_2 \in \text{Gen}(w_2)$ satisfying the induction hypothesis. The automaton $A' = A_1 \circ A_2$ (with \circ as defined in Lemma 3) is a valid epsilon-generalization. So $A \in \text{Gen}(w)$. By Lemma 3, $A' = A_1 \circ A_2$ recognizes $\mathcal{L}(A_1) \cdot \mathcal{L}(A_2) = \mathcal{L}(e_1) \cdot \mathcal{L}(e_2)$. Therefore, $\mathcal{L}(A) = \mathcal{L}(e)$. As the final state of A_2 had no outgoing

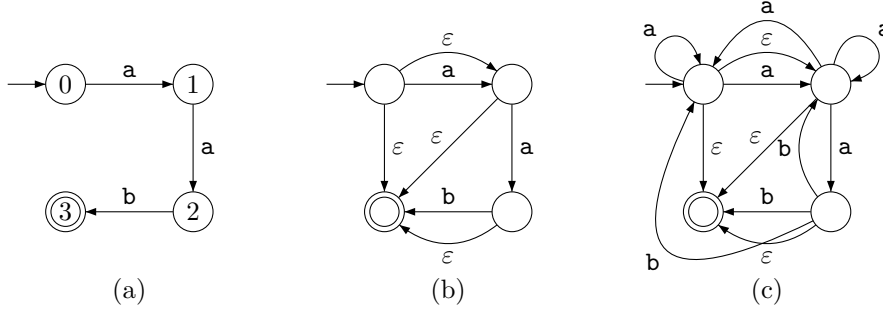


Figure 3: Construction of $\text{refine}_\epsilon(w, L)$, where $w = aab$ and $L = \{a^i b^{i+1} \mid i \geq 0\}$: (a) Initial automaton for w ; (b) the automaton after adding forward ϵ -transitions; (c) the final epsilon-generalization.

transitions, and we have not added any outgoing transitions from q_n , the final state of A' has no outgoing transitions.

As we can construct epsilon-generalizations for trivial star generalizations (depth $k = 1$), epsilon-generalizations of size k can be constructed from star generalizations of size $k - 1$, any element of $\Xi(w)$ must correspond to an equivalent element of $\text{Gen}(w)$. \square

We can incrementally construct an epsilon-generalization with respect to some co-context-free language \bar{L} by adapting algorithms for the intersection of context-free languages and finite automata, such as the *pre** algorithm described in [6].

Essentially, we maintain a table $\tau \subseteq Q \times \Gamma \times Q$ such that $(q_i, P, q_j) \in \tau$ iff the context-free production P can be generated by some sub-word recognized by a path from q_i to q_j . Whenever a new transition is added to the generalization, we update τ with any newly feasible productions; if the start production S is ever generated on a path from q_0 to q_n , the generalization has ceased to be valid – in which case we revert the table and discard the most recent augmentation. The *pre** algorithm, as described in [7], exhibits $O(|\Gamma||Q|^3)$ worst-case time complexity. In the worst case, where every generalization step fails after the maximum number of steps, this gives the generalization procedure a worst-case complexity of $O(|\Gamma||Q|^5)$.

Example 3. Consider again the star-generalization of aab described in Example 2. If we were instead constructing an epsilon-generalization, we start with the automaton A recognizing w , shown in Figure 3(a).

We first try greedily adding forwards epsilon transitions. This yields the automaton shown in Figure 3(b), corresponding to the language $(a?(ab?)?)^*$.

The first backward transition we attempt is (q_{1-1}, a, q_0) , followed by (q_{2-1}, a, q_1) . The next transition, (q_{3-1}, b, q_2) , cannot be added as it would accept abb , which is in L . This process continues, resulting in the final automaton shown in Figure 3(c). The language recognized by this automaton is $(a^*ab)^*a^*$, which is equivalent to the language obtained by star-generalization in Example 2.

3.3. Maximum generalization

The procedure described in the previous section constructs *some* maximal element of $\Xi_L(w)$ (or $\text{Gen}_L(w)$). It is, however, undirected; the generalization is chosen blindly from the set of possible maximal generalizations.

Even if the query languages are regularly separable, it is possible that the refinement step may choose an infinite sequence of generalizations which, though maximal, cannot separate the queries.

We can instead construct a generalization $\text{gen}(L, w)$ which computes the *union* of all maximal star-generalizations of w with respect to \bar{L} . That is, it computes $\xi_{\bar{L}}(w) = \bigcup \Xi_{\bar{L}}(w)$ directly. We shall refer to this as the *maximum* star-generalization. A possible (though inefficient) method for computing this is given in Figure 4.

```

maxgen( $L, w$ )
1:   let  $w$  be  $x_1 \cdot x_2 \cdots x_n$ 
2:    $S := \emptyset$ 
3:    $P := \{(i, j) \mid i, j \in [0, n], i \leq j\}$ 
4:   return gen( $L, \langle w, S \rangle, P$ )

maxgen( $L, \langle w, S \rangle, \emptyset$ )
5:   return  $\mathcal{L}(\langle w, S \rangle)$ 

maxgen( $L, \langle w, S \rangle, \{(i, j)\} \cup P$ )
6:    $R_f := \mathbf{gen}(L, \langle w, S \rangle, P)$ 
7:    $S' := S \cup \{(i, j)\}$ 
8:   if ( $L \cap \mathcal{L}(\langle w, S' \rangle) = \emptyset$ )
9:      $P' := \left\{ (i', j') \mid \begin{array}{l} (i', j') \in P \\ \wedge \quad (j \leq i' \vee j' \leq j) \\ \wedge \quad (j' \leq i \vee j \leq j') \end{array} \right\}$ 
10:     $R_f := R_f \cup \mathbf{gen}(L, \langle w, S' \rangle, P')$ 
11:  return  $R_f$ 

```

Figure 4: Computing $\xi_{\overline{L}}(w)$, the maximum star-generalization of w with respect to \overline{L} .

The procedure **maxgen** carries around S , a partial generalization, and P , the set of candidate $*$ -augmentations. At each stage, an augmentation e is selected from P , and we recursively compute the set of valid further generalizations of $\langle w, S \rangle$ both including and excluding e , finally taking the union of the sub-languages.

The maximum epsilon-generalization with respect to L – denoted by $\mathbf{g}_L(w)$ – may be constructed by an analogous procedure.

We now show that this generalization procedure is sufficiently powerful as to prove separability for any pair of regularly separable languages.

Lemma 4. *Consider a context-free language G , and regular language R such that $G \cap R = \emptyset$. Then for any word $w \in R$, there is some $e' \in \kappa(R)$ such that $\mathcal{L}(e') \subseteq \xi_{\overline{G}}(w)$.*

Proof. By Lemma 2, there is some $e \in \Xi(w) \cap \kappa(R)$. As $R \cap G = \emptyset$, we have $e \in \Xi_{\overline{G}}(w)$. Therefore $\mathcal{L}(e) \subseteq \bigcup \{\mathcal{L}(e') \mid e' \in \Xi_{\overline{G}}(w)\} = \xi_{\overline{G}}(w)$. \square

Corollary 1. *Consider a context-free language G , and regular language R such that $G \cap R = \emptyset$. Then for any word $w \in R$, there is some $e' \in \kappa(R)$ such that $\mathcal{L}(e') \subseteq \mathbf{g}_{\overline{G}}(w)$.*

Proof. This follows immediately from Lemma 4 and Theorem 3. \square

Theorem 4. *Given a pair of regularly separable context-free languages (L, L') and initial regular approximations R_L and $R_{L'}$ with $L \subseteq R_L$ and $L' \subseteq R_{L'}$, the refinement process described in Section 3 will construct a separating pair $(S_L, S_{L'})$ in a finite number of steps when refining using the maximum star- or epsilon-generalization.*

Proof. Consider the (unknown) regular language S separating L and L' . Assume $\kappa(S)$ and $\kappa(\overline{S})$ are as given in Theorem 2. Let K^i denote the elements of $\kappa(S) \cup \kappa(\overline{S})$ having non-empty intersection with the current approximation $R_L^i \cap R_{L'}^i$.

Assume that, at a given step, there is some word $w \in R_L^i \cap R_{L'}^i$. w must be in exactly one of S and \overline{S} ; we assume w is in S (the case of \overline{S} is symmetric). As $w \in S$, there must be some $r \in \kappa(S)$ such that $\mathcal{L}(r) \subseteq \xi_{\overline{L'}}(w) \subseteq \mathbf{g}_{\overline{L'}}(w)$. As $w \in R_L^i \cap R_{L'}^i$, and $w \in r$, we have $r \in K^i$.

As $R_{L'}^{i+1} = R_{L'}^i \setminus \xi_{\overline{L'}}(w)$ (or $R_{L'}^i \setminus \mathbf{g}_{\overline{L'}}(w)$), we have $\mathcal{L}(r) \cap R_{L'}^{i+1} = \emptyset$. Therefore, $K^{i+1} \subset K^i$. As $[K^1, K^2, \dots]$ is a decreasing sequence, and K^1 is finite, the refinement process must terminate after finitely many steps. \square

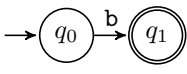
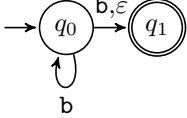
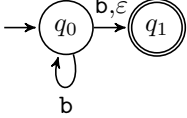
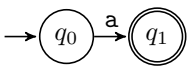
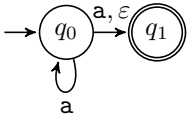
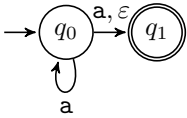
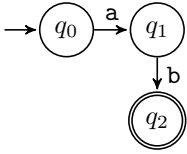
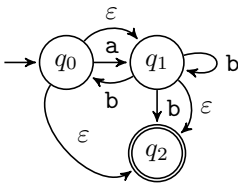
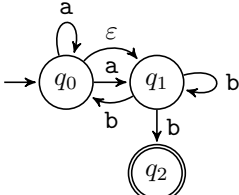
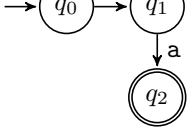
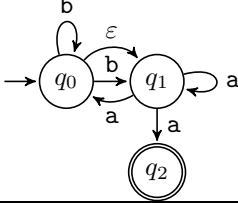
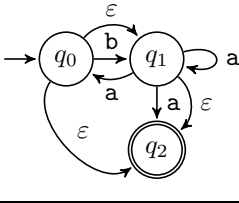
witness w	$\text{refine}_\varepsilon(G_1, w)$	$\text{refine}_\varepsilon(G_2, w)$
ε	ε	ε
		
		
		
		

Figure 5: Relevant witnesses and generalizations obtained by greedy epsilon generalization.

4. Example

Consider the two context-free grammars $G_1 = \langle \{S_1, A_1, B_1\}, \Sigma, P_1, S_1 \rangle$ and $G_2 = \langle \{S_2, A_2, B_2\}, \Sigma, P_2, S_2 \rangle$ where $\Sigma = \{a, b\}$ and P_1 and P_2 are, respectively,

$$\begin{array}{ll}
S_1 & \rightarrow A_1 B_1 \\
A_1 & \rightarrow aa \mid bb \mid aS_1 a \mid bS_1 b \\
B_1 & \rightarrow abB_1 \mid ab
\end{array}
\qquad
\begin{array}{ll}
S_2 & \rightarrow A_2 B_2 \\
A_2 & \rightarrow aa \mid bb \mid aS_2 a \mid bS_2 b \\
B_2 & \rightarrow baB_2 \mid ba
\end{array}$$

Note that $\mathcal{L}(G_1) = \{ww^R(\mathbf{ab})^+ \mid w \in \Sigma^*\}$ and $\mathcal{L}(G_2) = \{ww^R(\mathbf{ba})^+ \mid w \in \Sigma^*\}$.

The first step of our method will approximate G_1 and G_2 with finite-state automata A_1 and A_2 . The only requirement is that $\mathcal{L}(G_1) \subseteq \mathcal{L}(A_1)$ and $\mathcal{L}(G_2) \subseteq \mathcal{L}(A_2)$. For simplicity, assume $A_1 = A_2 = \Sigma^*$. Next, we check if $\mathcal{L}(A_1) \cap \mathcal{L}(A_2) \neq \emptyset$. In this case, the intersection is trivially not empty. Furthermore, our regular solver provides the witness $w = \varepsilon$. This cannot be generalized, so we eliminate ε from both approximations and try again, this time obtaining $w = \mathbf{b}$. In the third step we refine the regular approximations. We assume the use of greedy epsilon refinement, preferring backwards transitions. We first generalize the witness by calling $\text{refine}_\varepsilon(G_1, w)$ and $\text{refine}_\varepsilon(G_2, w)$ to produce new approximations $\mathcal{L}(A'_1) = \mathcal{L}(A_1) \setminus \text{refine}_\varepsilon(G_1, w)$ and $\mathcal{L}(A'_2) = \mathcal{L}(A_2) \setminus \text{refine}_\varepsilon(G_2, w)$, respectively. We show the automata obtained from $\text{refine}_\varepsilon(G_1, b)$ and $\text{refine}_\varepsilon(G_2, b)$ on the first row in Figure 5. In both cases, we obtain the language \mathbf{b}^* .

Since G_1 and G_2 are regularly separable, using maximal refinement would be guaranteed to eventually eventually halt proving that the languages are disjoint. While we have no such guarantee for greedy refinement, in this case it successfully proves separation after 5 refinement steps. Figure 5 depicts the rest of witnesses obtained as well as their generalizations produced by the procedure $\text{refine}_\varepsilon$.

5. Previous Refinement Techniques

Several CEGAR-based approaches have been proposed for testing intersection of context-free languages. In this section, we attempt to characterise the expressiveness of existing refinement methods. For these comparisons we do not consider the effect of initial regular approximations, as they do not affect the expressiveness of the refinement method. For any fixed finite set of regularly-separable languages, there is always some approximation scheme which allows the languages to be trivially proven separate; however it is impossible to define such an approximation in general.

The idea of using CEGAR to check the intersection of CFGs was pioneered by Bouajjani *et al.* [2] for the context of verifying concurrent programs with recursive procedures. They rely on the concept of *refinable finite-chain abstraction* consisting of computing the series $(\alpha_i)_{i \geq 1}$ which overapproximates the language of a CFG L (i.e., $L \subseteq \alpha_i(L)$) such that $\alpha_1(L) \supset \alpha_2(L) \supset \dots \supseteq L$. The method is parameterized by the refinable abstraction. [2] describe several possible abstractions but no experimental evaluation is provided. Chaki *et al.* [4] extend [2] by, among other contributions, implementing and evaluating this method. The experimental evaluation of Chaki *et al.* uses both the i^{th} -prefix and i^{th} -suffix abstractions. Given language L , the i^{th} -prefix abstraction $\alpha_i(L)$ is the set of words of L of length less than i , together with the set of prefixes of length i of L . The i^{th} -suffix abstraction can be defined analogously. We next provide a theorem about the expressiveness of the i^{th} -prefix abstraction. A similar result holds for the i^{th} -suffix abstraction.

Theorem 5. *There exist regularly separable languages that cannot be proven separate by the i^{th} -prefix abstraction.*

Proof. Consider the languages $R_1 = \mathbf{a}^*\mathbf{b}$, $R_2 = \mathbf{a}^*\mathbf{c}$. $R_1 \cap R_2$ is empty. However, for a given length i , the string \mathbf{a}^i forms a prefix to words in both R_1 and R_2 . It follows that the intersection of the two abstractions will always be non-empty, so the refinement method cannot prove the languages separate. \square

The LCEGAR method described by Long *et al.* [13] is based on a similar refinement framework, but the approach differs radically. They maintain a pair of context-free grammars A_1, A_2 over-approximating the intersection of the original languages. At each refinement step, an *elementary bounded language* B_i is generated from each grammar A_i .² The refinement ensures $B_i \cap A_i \neq \emptyset$, but B_i is not necessarily either an over- or under-approximation of A_i . They then compute $I = B_i \cap L_1 \cap L_2$. If I is non-empty, $L_1 \cap L_2$ must also be non-empty. If I is empty, then the approximations can safely be refined by subtracting the B_i .

We now wish to characterise the set of languages for which LCEGAR can prove separation. Note that we do not consider the initial approximation; for any fixed pair of regularly-separable languages, there necessarily exists *some* approximation method which immediately proves separation without refinement.

Theorem 6. *There exist non-regularly-separable languages which can be proven separate by LCEGAR.*

Proof. Consider the languages L and \overline{L} , where $L = \{\mathbf{a}^n\mathbf{b}^n \mid n \geq 0\}$. These are not regularly separable. Still, LCEGAR will find that they do not overlap. Assume initial approximations $A_1 = L_1$ and $A_2 = L_2$. At the first iteration, LCEGAR may choose bounded approximation $B = \mathbf{a}^*\mathbf{b}^*$. It will find $B \cap L_1 \cap L_2 = \emptyset$, then update $A_1 = A_1 \setminus B = \emptyset$. As $A_1 = \emptyset$, the refinement process has successfully proven separation. \square

Lemma 5. *For any bounded regular language $B = w_1^* \dots w_k^*$, there is some word p that is not a substring of any word in B .*

Proof. For each word w_1 , we pick some character c_1 which differs from the *last* character of w_1 . We then construct $p = p_1 \dots p_k$, such that:

$$p_i = \underbrace{c_i \dots c_i}_{|w_i|}$$

² An elementary bounded language is some language of the form $B = w_1^* \dots w_k^*$, where each w_i is a (finite) word in Σ^* .

Assume there is some word $t = up_1 \dots p_kv \in B$. t must consist of some number of occurrences of w_1 through w_k , in order. Since p_1 differs from the last character of w_1 , up_1 cannot consist only of occurrences of w_1 ; therefore, $p_2 \dots p_k$ must be made up of occurrences of w_2 through w_k .

Similarly, since no occurrence of w_2 may end in p_2 , so $p_3 \dots p_k$ must consist only of w_3 through w_k . By induction, we find that p_k must be an occurrence of w_k . However, no occurrence of w_k may occur in p_k . Therefore, there can be no word $t \in B$ such that $t \in \Sigma^*p\Sigma^*$. \square

Corollary 2. *For any finite set of bounded regular languages $\{B_1, \dots, B_n\}$, we can construct some substring p that is not a substring of $B_1 \cup \dots \cup B_n$.*

Proof. By Lemma 5, we can find p_1, \dots, p_n such that p_i is not a substring in B_i . Then $p = p_1 \dots p_n$ cannot occur as a substring in $B_1 \cup \dots \cup B_n$. \square

Theorem 7. *There exist regularly separable languages for which the LCEGAR refinement method cannot prove separability.*

Proof. Consider an LCEGAR process with $L_1 = A_1 = (a|b)^*a$, and $L_2 = A_2 = (a|b)^*b$. These languages are disjoint, and regularly separable. After some finite number of steps, the approximations have been refined with bounded languages $\{B_1, \dots, B_n\}$. By Corollary 2, there is some substring p such that $\Sigma^*p\Sigma^* \subseteq (\overline{B_1} \cap \dots \cap \overline{B_n})$. The updated approximation A'_1 is non-empty, as it contains pa . Similarly, the approximation A'_2 is non-empty, as it contains pb .

Since after any finite sequence of refinement steps neither A'_1 nor A'_2 is empty, the refinement process will never prove separation of L_1 and L_2 . \square

From Theorems 4, 6 and 7, we conclude that the classes of languages which can be proven separate by LCEGAR and COVENANT are incomparable.

6. Experimental Evaluation

We have implemented the CEGAR method proposed in this paper in a prototype tool called COVENANT³. The tool is implemented in C++ and parameterized by the initial approximation and the refinement procedure. COVENANT implements the method described in [16] for approximating CFGs with strongly regular languages as well as the coarsest abstraction Σ^* for comparison purposes. For refinement, the tool implements both the greedy and maximum star-epsilon generalizations (described in Sections 3.1 and 3.3, respectively). COVENANT currently implements only the classical product construction for solving the intersection of regular languages but other regular solvers (*e.g.*, [8, 10]) can be easily integrated⁴.

To assess the effectiveness of our tool, we have conducted two experiments. First, we used COVENANT for proving safety properties in recursive multi-threaded programs. Second, we crafted pairs of challenging context free grammars and intersected them using COVENANT. The motivation for this second experiment was to exercise features of COVENANT that were not required during the first experiment. All experiments were run on a single core of a 2.4GHz Core i5-M520 with 7.8Gb memory.

Safety verification of recursive multi-threaded programs. Bouajjani *et al.* [2] was pioneered showing that the safety verification problem of recursive multi-threaded programs can be reduced to check whether the intersection of context free languages is empty. Since then, several encodings have been described [2, 4, 13]. As a result, we can use COVENANT to prove certain safety properties in recursive multi-threaded programs assuming the programs have been translated accordingly. We briefly exemplify the translation of a concurrent program to context-free grammars, using the approach of [13].

For simplicity, we assume a concurrency model in which communication is based on shared memory. Shared memory is modelled via a set of global variables. We assume that each statement is executed

³Publicly available at <https://bitbucket.org/jorgenavas/covenant> together with all the benchmarks used in this section.

⁴In fact, an initial implementation of COVENANT was tested using REVENANT [8], an efficient regular solver based on bounded model checking with interpolation, though the released version does not incorporate it.

<pre> x = 0; y = 0; p1 () { n0: x = not y ; n1: if(*) p1(); n2: x = not y ; n3: } p2 () { m0: y = not x ; m1: if(*) p2(); m2: y = not x ; m3: } if (x and y) error(); </pre>	<p>CFG₁</p> <p>// control flow of thread p1</p> $N_0 \rightarrow \llbracket x = \text{not } y \rrbracket N_1$ $N_1 \rightarrow N_0 N_2 \mid N_2$ $N_2 \rightarrow \llbracket x = \text{not } y \rrbracket N_3$ $N_3 \rightarrow \llbracket x \rrbracket$ <p>//encoding of instructions for p1</p> $\llbracket x = \text{not } y \rrbracket \rightarrow S_{p_2} \text{ y_at_0 set_x_1 } S_{p_2} \mid S_{p_2} \text{ y_at_1 set_x_0 } S_{p_2}$ $\llbracket x \rrbracket \rightarrow \text{x_at_1}$ <p>//synchronization with p2's actions</p> $S_{p_2} \rightarrow \text{x_at_0 } S_{p_2} \mid \text{x_at_1 } S_{p_2} \mid \text{set_y_0 } S_{p_2} \mid \text{set_y_1 } S_{p_2} \mid \varepsilon$	<p>CFG₃</p> <p>//Modelling variable x</p> $X_{false} \rightarrow \text{x_at_0 } X_{false} \mid \text{set_x_0 } X_{false} \mid \text{set_x_1 } X_{true} \mid S_x X_{true} \mid \varepsilon$ $X_{true} \rightarrow \text{x_at_1 } X_{true} \mid \text{set_x_1 } X_{true} \mid \text{set_x_0 } X_{false} \mid S_x X_{true} \mid \varepsilon$ <p>//Synchronization with y</p> $S_x \rightarrow \text{y_at_0 } S_x \mid \text{set_y_0 } S_x \mid \text{y_at_1 } S_x \mid \text{set_y_1 } S_x \mid \varepsilon$
	<p>CFG₂</p> <p>//Control flow of thread p2</p> $M_0 \rightarrow \llbracket y = \text{not } x \rrbracket M_1$ $M_1 \rightarrow M_0 M_2 \mid M_2$ $M_2 \rightarrow \llbracket y = \text{not } x \rrbracket M_3$ $M_3 \rightarrow \llbracket y \rrbracket$ <p>//Encoding of instructions for p2</p> $\llbracket y = \text{not } x \rrbracket \rightarrow S_{p_1} \text{ x_at_0 set_y_1 } S_{p_1} \mid S_{p_1} \text{ x_at_1 set_y_0 } S_{p_1}$ $\llbracket y \rrbracket \rightarrow \text{y_at_1}$ <p>//Synchronization with p1's actions</p> $S_{p_1} \rightarrow \text{y_at_0 } S_{p_1} \mid \text{y_at_1 } S_{p_1} \mid \text{set_x_0 } S_{p_1} \mid \text{set_x_1 } S_{p_1} \mid \varepsilon$	<p>CFG₄</p> <p>//Modelling variable y</p> $Y_{false} \rightarrow \text{y_at_0 } Y_{false} \mid \text{set_y_0 } Y_{false} \mid \text{set_y_1 } Y_{true} \mid S_y Y_{true} \mid \varepsilon$ $Y_{true} \rightarrow \text{y_at_1 } Y_{true} \mid \text{set_y_1 } Y_{true} \mid \text{set_y_0 } Y_{false} \mid S_y Y_{true} \mid \varepsilon$ <p>//Synchronization with x</p> $S_y \rightarrow \text{x_at_0 } S_y \mid \text{set_x_0 } S_y \mid \text{x_at_1 } S_y \mid \text{set_x_1 } S_y \mid \varepsilon$

Figure 6: A concurrent Boolean program (SharedMem) and its translation to CFGs.

atomically. We will consider only *Boolean programs*. Any program P can be translated into a Boolean program $\mathcal{B}(P)$ using techniques such as predicate abstraction [9]. A key property is that $\mathcal{B}(P)$ is an over-approximation of P preserving the control flow of P but the only type available in $\mathcal{B}(P)$ is Boolean. Therefore, if $\mathcal{B}(P)$ is correct then P must be correct but, of course, if $\mathcal{B}(P)$ is unsafe P may be still safe. Each (possibly recursive) procedure in $\mathcal{B}(P)$ is modelled as a context-free grammar as well as each shared variable specifying the possible values that the variable can take. In addition, extra production rules are added to specify the synchronization points.

The left hand column of Figure 6 shows a small program **SharedMem** [13]. It consists of two symmetric, recursive procedures $p1$ and $p2$ which are executed by two different threads. The communication between the threads is done through the global variables x and y which are initially set to 0. Note that the program is already Boolean since x and y can only take values 0 and 1. We would like to prove that after $p1$ and $p2$ terminate, x and y cannot be true simultaneously.

The rest of Figure 6 describes the corresponding translation to context-free grammars. The four resulting

grammars, which we explain shortly, are

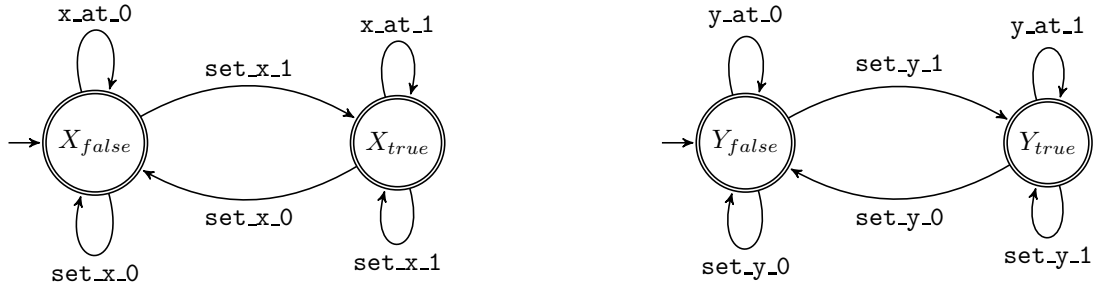
$$\begin{aligned}
\text{CFG}_1 &: \langle \{N_0, N_1, N_2, N_3, \llbracket x = \text{not } y \rrbracket, \llbracket x \rrbracket, S_{p_2}\}, \Sigma, P_1, N_0 \rangle \\
\text{CFG}_2 &: \langle \{M_0, M_1, M_2, M_3, \llbracket y = \text{not } x \rrbracket, \llbracket y \rrbracket, S_{p_1}\}, \Sigma, P_2, M_0 \rangle \\
\text{CFG}_3 &: \langle \{X_{false}, X_{true}, S_x\}, \Sigma, P_3, X_{false} \rangle \\
\text{CFG}_4 &: \langle \{Y_{false}, Y_{true}, S_y\}, \Sigma, P_4, Y_{false} \rangle
\end{aligned}$$

where $\Sigma = \{\text{x_at_0}, \text{x_at_1}, \text{y_at_0}, \text{y_at_1}, \text{set_x_0}, \text{set_x_1}, \text{set_y_0}, \text{set_y_1}\}$ and P_1, P_2, P_3 , and P_4 are the respective sets of productions, as shown in Figure 6.

Procedures p_1 and p_2 are translated into CFG_1 and CFG_2 , respectively. First, we need to encode the control flow of the procedures. For instance, “ p_1 reaches location n_0 and it executes the statement $x = \text{not } y$ ” is translated into the grammar production $N_0 \rightarrow \llbracket x = \text{not } y \rrbracket N_1$, where N_1 represents the next program location n_1 . We use the notation $\llbracket s \rrbracket \in V$ to refer to the corresponding translation of statement s . A function call such as “ p_1 calls itself recursively after location n_1 is executed” is translated through the production $N_1 \rightarrow N_0 N_2$ where N_0 is the entry location of the callee function and N_2 is the continuation of the caller after the callee returns. The non-terminal symbol $\llbracket x = \text{not } y \rrbracket$ models the execution of negating y and storing its result in x . We create a terminal symbol for each possible action on x (and analogously for y): x_at_0 (the value of x is 0), x_at_1 (the value of x is 1), set_x_0 (x is updated to 0), and set_x_1 (x is updated to 1). For instance, the grammar production $\llbracket x = \text{not } y \rrbracket \rightarrow S_{p_2} \text{y_at_0 set_x_1 } S_{p_2}$ represents that if we read 0 as the value of y then it must be followed by writing 1 to x . The rest of logical operations are encoded similarly.

Note that whenever a global variable is read or written we need to consider the synchronization between threads. For this purpose, we define the non-terminal symbols S_{p_2} (S_{p_1}) which loops zero or more times with all possible actions of p_2 (p_1): value of x is 0 (value of y is 0), value of x is 1 (value of y is 1), y is updated to 0 (x is updated to 0), and y is updated to 1 (x is updated to 1).

Next, we need to model which are the possible values that x and y can take. For this we use CFG_3 and CFG_4 , respectively. Ignoring synchronization, the set of values that x and y can take are indeed expressed by regular automata:



Finally, we need to synchronize x and y by allowing them to loop zero or more times while new values from the other variable can be generated. We use non-terminal symbols (and their productions) S_x and S_y for that.

Once we have obtained the CFGs described in Figure 6 we are ready to ask reachability questions. For this example, we would like to prove that when threads start at n_0 and m_0 , respectively, **error** cannot be reachable simultaneously by both threads. This question can be answered by checking if the intersection of the above CFGs is empty. If the intersection is not empty then COVENANT will return a witness $w \in \Sigma^*$ containing the sequence of reads and writes to x and y . Otherwise, COVENANT will return either “yes” (that is, the program is safe) if the languages of the CFGs are regularly separable or run until resources are exhausted.

We have tested COVENANT with the programs used in [13] and compared with LCEGAR [13]. There are two classes of programs: textbook Erlang programs and several variants of a real Bluetooth driver. The Bluetooth variants labelled W/ Heuri are encoded with an unsound heuristic that permits context switches

Program		COVENANT		LCEGAR	
				PDC	CB
SharedMem	safe	0.01		14.37	24.75
Mutex	safe	0.04		6.12	0.14
RA	safe	0.01		∞	0.39
Modified RA	safe	0.03		∞	27.90
TNA	unsafe	0.01		0.02	0.25
Banking	unsafe	0.01		∞	3.36

(a) Verification of multi-thread Erlang programs

Program		COVENANT		LCEGAR	
				PDC	CB
Version 1	unsafe	0.84		19.74	21.04
Version 2	unsafe	0.25		5560.00	4852.00
Version 2 w/ Heuri	unsafe	0.11		44.68	38.89
Version 3 (1A2S)	unsafe	0.12		217.74	217.27
Version 3 (1A2S) w/ Heuri	unsafe	0.05		6.68	11.37
Version 3 (2A1S)	safe	0.27		4185.00	3981.00

(b) Verification of multi-thread Bluetooth drivers

		COVENANT				LCEGAR	
		Σ^*		[16]		PDC	CB
		Greedy	Gen	Greedy	Gen		
$C_1 \cap C_7$	sat	8 (0.01)	11 (7.88)	5 (0.01)	8 (6.20)	∞	–
$C_7 \cap C_1$						0 (0.13)	0 (0.32)
$C_1 \cap C_8$	sat	8 (0.01)	13 (8.36)	7 (0.01)	9 (2.22)	0 (20.28)	–
$C_8 \cap C_1$						∞	∞
$C_2 \cap C_3$	sat	10 (0.01)	13 (9.10)	2 (0.01)	2 (0.02)	0 (0.03)	0 (0.01)
$C_3 \cap C_2$						0 (0.03)	0 (0.01)
$C_2 \cap C_4$	unsat	15 (0.02)	∞	3 (0.01)	3 (0.80)	1 (0.01)	0 (0.01)
$C_4 \cap C_2$						∞	0 (0.01)
$C_3 \cap C_4$	unsat	11 (0.01)	∞	2 (0.01)	2 (0.04)	0 (0.01)	0 (0.01)
$C_4 \cap C_3$						0 (0.01)	0 (0.01)
$C_5 \cap C_6$	unsat	6 (0.01)	∞	5 (0.01)	∞	∞	0 (0.01)
$C_6 \cap C_5$						∞	0 (0.01)
$C_5 \cap C_7$	sat	14 (0.04)	∞	11 (0.02)	∞	∞	–
$C_7 \cap C_5$						0 (0.33)	∞
$C_5 \cap C_8$	sat	7 (0.01)	9 (2.81)	5 (0.01)	5 (3.54)	∞	–
$C_8 \cap C_5$						0 (0.04)	∞
$C_6 \cap C_7$	sat	14 (0.04)	∞	11 (0.02)	∞	∞	–
$C_7 \cap C_6$						0 (0.10)	∞
$C_6 \cap C_8$	sat	8 (0.01)	9 (2.86)	5 (0.01)	5 (3.46)	0 (1.21)	–
$C_8 \cap C_6$						∞	∞
$C_7 \cap C_8$	sat	4 (0.01)	4 (0.01)	3 (0.01)	3 (0.01)	0 (0.70)	–
$C_8 \cap C_7$						∞	–

(c) Interesting/challenging grammars (∞ indicates time-out at 60 sec and “–” a raised exception.)

Table 1: Comparison of COVENANT with LCEGAR, on several classes of context free grammars; times in seconds.

only at basic block boundaries. We refer readers to Appendix C for a detailed description of the programs as well as the safety properties.

Table 1(a) and Table 1(b) show the times in seconds for both solvers when proving the Erlang programs and the Bluetooth drivers. The symbol ∞ indicates that the solver failed to terminate after 2 hours. We ran LCEGAR using the settings suggested by the authors and tried with the two available initial abstractions: *pseudo-downward closure* (PDC) and *cycle breaking* (CB). For our tool, we used as the initial abstraction the one described in [16] which is described in Appendix A. We also tried Σ^* but COVENANT did not converge for any of the programs in a reasonable amount of time.

It is somewhat surprising that all properties were successfully proven by LCEGAR using the initial regular approximation, including Bluetooth instances. The same is true for COVENANT, except for **Version 1** which required 12 refinements using the greedy strategy. Nevertheless, these programs show cases in which COVENANT can significantly outperform LCEGAR. Since almost no refinements were required by any of the tools, it also suggests that the approximation of all CFGs at once and the use of a regular solver is often a more efficient choice than relying on computing intersection of CFLs and regular languages as LCEGAR does.

Other interesting CFLs. The verification instances [13] are in fact all solved with no use of refinement, by LCEGAR as well as by COVENANT (with the exception of one instance). To explore more interesting cases that exercise the refinement procedures, we have added experiments involving the following languages ($\Sigma = \{a, b\}^*$; note that C_5 is $\mathcal{L}(G_1)$ from Section 4 and C_6 is $\mathcal{L}(G_2)$):

$$\begin{array}{l|l} C_1 : \{ww^R \mid w \in \Sigma^*\} & C_5 : \{ww^R(\mathbf{ab})^+ \mid w \in \Sigma^*\} \\ C_2 : \{wcw^R \mid w \in \Sigma^*\} & C_6 : \{ww^R(\mathbf{ba})^+ \mid w \in \Sigma^*\} \\ C_3 : \{a^n \mathbf{ca}^n \mid n > 0\} & C_7 : \{w \in \Sigma^* \mid w \text{ has equal numbers of as and bs}\} \\ C_4 : \{a^n \mathbf{cb}^n \mid n > 0\} & C_8 : \{ww' \mid |w| = |w'|, w \neq w'\} \end{array}$$

Table 1(c) shows the pairs of languages whose disjointness can be proven or a counterexample can be found requiring at least one refinement for COVENANT. We ignore pairs of languages which are disjoint but not regularly separable.

We ran COVENANT using two initial abstractions: Σ^* and the more precise one described by Nederhof [16]. For each one, we used our greedy (Greedy) refinement described in Section 3.1 and the complete refinement (Gen) from Section 3.3. We compared again with LCEGAR using its two abstractions PDC and CB. For a given $C_i \cap C_j$ LCEGAR checks first whether $\mathcal{L}(C_i) \cap \mathcal{L}(\alpha(C_j)) = \emptyset$ and then, only if it is not empty a refinement is triggered. Therefore, LCEGAR fixes a priori that the first input grammar will not be abstracted while the second will (that is, the order in which the grammars are given to LCEGAR matters). That is why we test both $C_i \cap C_j$ and $C_j \cap C_i$. In COVENANT the order is irrelevant. We use the format #R (T) to indicate that the tool needed #R refinements to prove disjointness or to find a counterexample, in T seconds. We set a timeout (∞) of 60 seconds.

Table 1(c) indicates that, generally, the more precise the initial abstraction, the fewer refinements are necessary. This claim was also made in [13] although we were not able to fully confirm it because LCEGAR raised an exception with many of the instances while using CB (denoted by the symbol $-$). Interestingly, Greedy performs quite well, terminating for all instances. This suggests that Greedy might be a good practical choice in cases where Gen spends too much time computing the generalization of the witnesses.

Regarding LCEGAR either a timeout is reached or the tool can either prove disjointness or find a witness without any refinement except for one instance ($C_2 \cap C_4$). It is worth noticing that even if both tools would start with the same initial abstraction LCEGAR might not refine at all while COVENANT might do. The reason is that LCEGAR does not abstract all the CFLs which forced us try with both pair orderings. On the other hand, this gives some unpredictability to LCEGAR because depending on the ordering, the tool can behave very differently (for example, $(C_2 \cap C_4)$ versus $(C_4 \cap C_2)$).

7. Conclusions and Future Work

We have presented a CEGAR-based semi-decision procedure for regular separability of context-free languages. We have described two refinement strategies; an inexpensive greedy approach, and a more expensive

exhaustive strategy. We have implemented these approaches in a prototype solver, COVENANT. The method outperforms LCEGAR on a range of verification and language-theoretic instances. The greedy approach often requires more refinement steps, but tends to quickly find witnesses in cases with non-empty intersections; the exhaustive method performs substantially more expensive refinement steps, but can prove separation of some instances not solved by other methods.

The maximum ε -generalization algorithm can become extremely expensive for large witnesses. It would be fruitful to consider whether we can find a cheaper generalization which still ensures completeness. Similarly, it may be possible to develop a specialized intersection algorithm for computing ε -generalizations, rather than relying on the standard regular/context-free intersection algorithm.

Visibly pushdown languages (VPLs) [1] have become popular and possess closure and decidability properties very similar to those of the class of regular languages. It would be interesting to explore algorithms for approximation by VPLs. (Of the languages C_1 – C_8 in Section 6, only C_4 is a VPL.)

Acknowledgments

We wish to thank Georgel Calin for providing the test programs and the implementation of LCEGAR. We also thank Pierre Ganty for fruitful discussions about this topic. We acknowledge support of the Australian Research Council through Discovery Project Grant DP140102194.

References

- [1] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the 36th Annual ACM Symposium on the Theory of Computing*, pages 202–211. ACM Publ., 2004.
- [2] Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Proceedings of the 30th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 62–73. ACM Publ., 2003.
- [3] Janusz A. Brzozowski and Rina S. Cohen. On decompositions of regular events. *Journal of the ACM*, 16(1):132–144, 1969.
- [4] S. Chaki, E. Clarke, N. Kidd, T. Reps, and T. Touili. Verifying concurrent message-passing C programs with recursive calls. In H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 334–349. Springer, 2006.
- [5] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.
- [6] Javier Esparza and Peter Rossmanith. An automata approach to some problems on context-free grammars. In C. Freksa, M. Jantzen, and R. Valk, editors, *Foundations of Computer Science: Potential, Theory, Cognition*, volume 1337 of *Lecture Notes in Computer Science*, pages 143–152. Springer, 1997.
- [7] Javier Esparza, Peter Rossmanith, and Stefan Schwoon. A uniform framework for problems on context-free grammars. *Bulletin of the EATCS*, 72:169–177, 2000.
- [8] Graeme Gange, Jorge A. Navas, Peter J. Stuckey, Harald Søndergaard, and Peter Schachte. Unbounded model-checking with interpolation for regular language constraints. In N. Piterman and S. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *Lecture Notes in Computer Science*, pages 277–291. Springer, 2013.
- [9] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.
- [10] Pieter Hooimeijer and Westley Weimer. StrSolve: Solving string constraints lazily. *Automated Software Engineering*, 19(4):531–559, 2012.
- [11] H. B. Hunt, III. On the decidability of grammar problems. *Journal of the ACM*, 29(2):429–447, 1982.
- [12] Nicholas Kidd. Bluetooth protocol. <http://pages.cs.wisc.edu/~kidd/bluetooth>.
- [13] Zhenyue Long, Georgel Calin, Rupak Majumdar, and Roland Meyer. Language-theoretic abstraction refinement. In J. de Lara and A. Zisman, editors, *Fundamental Approaches to Software Engineering*, volume 7212 of *Lecture Notes in Computer Science*, pages 362–376, 2012.
- [14] Benedek Nagy. A normal form for regular expressions. In *Supplemental Papers for the Eighth International Conference on Developments in Language Technology*, CDMTCS Research Report Series, pages 53–62. CDMTCS, 2004.
- [15] Mark-Jan Nederhof. Practical experiments with regular approximation of context-free languages. *Computational Linguistics*, 26(1):17–44, 2000.
- [16] Mark-Jan Nederhof. Regular approximation of CFLs: A grammatical view. In H. Bunt and A. Nijholt, editors, *Advances in Probabilistic and Other Parsing Technologies*, volume 16 of *Text, Speech and Language Technology*, pages 221–241. Springer, 2000.
- [17] Shaz Qadeer and Dinghao Wu. KISS: keep it simple and sequential. In *PLDI*, pages 14–24, 2004.
- [18] Thomas G. Szymanski and John H. Williams. Non-canonical parsing. In *Conference Record of the 14th Annual Symposium on Switching and Automata Theory*, pages 122–129. IEEE Comp. Soc., 1973.

Appendix A. Regular Abstractions of Context-Free Grammars

Nederhof [15, 16] proposed a transformation that converts a context-free grammar into a finite automaton. We present the algorithm NEDERHOFAPPROXIMATION in Figure A.7 that performs the whole transformation in two steps. The first, STRONGLYREGGRAMMAR, describes how to convert a context-free grammar into a strongly regular grammar. The second step, MAKEFA, builds a finite-state automaton from a given strongly regular grammar.

```

NEDERHOFAPPROXIMATION( $G$ )
  let  $SCC$  be the strongly connected components of  $G$ 
   $G' \equiv (V', \Sigma, P', S') := \text{STRONGLYREGGRAMMAR}(G, SCC)$ 
   $Q = \{q_0, q_F\}$  where  $q_0$  and  $q_F$  are new fresh states;
   $F := \{q_F\}; \Delta := \emptyset$ 
  MAKEFA( $q_0, S', q_F, G', SCC$ )
  return  $R \equiv (Q, \Sigma, \Delta, q_0, F)$ 

```

Figure A.7: Approximating a context-free grammar G with a finite automaton R such that $\mathcal{L}(G) \subseteq \mathcal{L}(R)$

Prior to describing the procedures STRONGLYREGGRAMMAR and MAKEFA we present some useful definitions. Consider partitions of the set of non-terminal symbols V . We define A and B to be part of the same partition if A and B are *mutually recursive*. That is, $A \Rightarrow^* \alpha B \beta$ and $B \Rightarrow^* \alpha' A \beta'$ for some sentential forms α, β, α' and β' .

Definition 10 (Left- and Right-Linearity). A production is *left-linear* iff it is of the form $A \rightarrow B w$ or $A \rightarrow w$, where $w \in \Sigma^*$. It is *right-linear* iff it is of the form $A \rightarrow w B$ or $A \rightarrow w$, where $w \in \Sigma^*$.

Definition 11 (Strongly regular grammars). A strongly regular grammar is a grammar in which the productions are either all left-linear or all right-linear.

Definition 12 (Left- and Right-Generating). A set of mutually recursive nonterminals S is left (right) generating if there exists a grammar production $A \rightarrow \alpha B \gamma, \alpha \neq \varepsilon$ ($A \rightarrow \alpha B \gamma, \gamma \neq \varepsilon$) and $A \in S$.

Assuming that we have a strongly regular grammar G we can classify each mutually recursive set S as⁵:

“left” if S is not left and right generating
 “right” if S is left and not right generating
 “cyclic” if S is neither left nor right generating

Figure A.8 shows the transformation, suggested by Nederhof, to convert an arbitrary context-free grammar to a strongly regular grammar. This transformation is based on the following observation. A context-free grammar consisting of productions of the form $A \rightarrow^* \alpha A \beta$ with both α, β non-empty might not be represented as a strongly regular grammar. The intuition is that α and β might be related through an “unbounded” communication (*i.e.* some correlation between the occurrences of a’s and b’s) no expressible by regular languages.

The procedure STRONGLYREGGRAMMAR takes a context-free grammar G and SCC , the set of strongly connected components that identify the set of mutually recursive nonterminal symbols of G . The algorithm identifies in a conservative manner situations where an unbounded communication can arise and breaks it by adding instead either left or right linear productions.

The procedure iterates over all strongly connected components in an arbitrary order and checks if all productions of the nonterminals in each component are either left or right linear. If yes, no transformation needs to be applied (line 19). Otherwise, it applies some transformations in order to convert all productions in the same strongly connected component into either left or right linear, as described at lines 7 and 12-17. The transformation assumes that nonterminals that do not belong to SCC_i are considered as terminals here,

⁵The case where S is both “left” and “right” is not applicable here since the grammar G is strongly regular

```

STRONGLYREGGRAMMAR( $\langle V, \Sigma, P, S \rangle, SCC$ )
1:  $V' := V; S' := S; P' := \emptyset$ 
2: foreach  $i \in \{0, \dots, |SCC|\}$ 
3:   if there exists a production with lhs in  $SCC_i$  neither left- nor right- linear
4:     foreach  $A \in SCC_i$  do
5:       create a fresh nonterminal  $A' \notin V$ 
6:        $V' := V' \cup \{A'\}$ 
7:        $P' := P' \cup \{A' \rightarrow \varepsilon\}$ 
8:       foreach  $A \rightarrow \alpha \in P$  with  $\alpha \in (\Sigma \cup (V \setminus SCC_i))^*$  do
9:          $P' := P' \cup \{A \rightarrow \alpha A'\}$ 
10:      foreach  $A \rightarrow \alpha_0 B_1 \alpha_1 B_2 \alpha_2 \dots B_m \alpha_m \in P$  with  $m \geq 0$ 
11:         $B_1, \dots, B_m \in SCC_i$ , and  $\alpha_0, \dots, \alpha_m \in (\Sigma \cup (V \setminus SCC_i))^*$  do
12:           $P' := P' \cup \{ A \rightarrow \alpha_0 B_1$ 
13:             $B'_1 \rightarrow \alpha_1 B_2$ 
14:             $B'_2 \rightarrow \alpha_2 B_3$ 
15:             $\dots$ 
16:             $B'_{m-1} \rightarrow \alpha_{m-1} B_m$ 
17:             $B'_m \rightarrow \alpha_m A' \}$ 
18:      else
19:        foreach  $A \in SCC_i$  and  $A \rightarrow X \in P$  do  $P' := P' \cup \{A \rightarrow X\}$ 
20: return  $(V', \Sigma, P', S')$ 

```

Figure A.8: Converting a context-free grammar into a strongly regular grammar

for determining if a production of SCC_i is right-linear or left-linear. This allows traversing the strongly connected components in any order. We show next how the transformation works through an example.

Example 4 (Conversion to strongly regular grammar). Let $G = (\{a, b, c\}, \{A, B\}, A, P)$, where P is the set of productions:

- (P1) $A \rightarrow aBb$
(P2) $A \rightarrow c$
(P3) $B \rightarrow A$

The set of strongly connected components is $\{\{A, B\}\}$. Therefore there is only one set $\{A, B\}$ of mutually recursive nonterminals. Line 7 adds to P' the rules:

- $A' \rightarrow \varepsilon$
 $B' \rightarrow \varepsilon$

From P1 and executing lines 12-17 we add to P' the rules:

- $A \rightarrow aB$
 $B' \rightarrow bA'$

From P2 and executing line 9 we add to P' the rule:

- $A \rightarrow cA'$

From P3 and executing lines 12-17 we add to P' the rule:

- $B \rightarrow A$
 $A' \rightarrow B'$

Finally, putting all rules of P' together and after some trivial simplifications:

- $A \rightarrow aA \mid cA'$
 $A' \rightarrow \varepsilon \mid bA'$

```

MAKEFA( $q_0, A, q_1, G \equiv (V, \Sigma, P, S), SCC$ )
Global variables: set of states  $Q$ , transition relation  $\Delta$ 

1: if  $A = \varepsilon$  then  $\Delta \leftarrow \Delta \cup \{(q_0, \varepsilon, q_1)\}$ 
2: else if  $A = a \in \Sigma$  then  $\Delta \leftarrow \Delta \cup \{(q_0, a, q_1)\}$ 
3: else if  $A = XY, X \in V, Y \in V^+$  then
4:    $Q = Q \cup \{q\}$  where  $q$  is a new fresh state;
5:   MAKEFA( $q_0, X, q, G, SCC$ );
6:   MAKEFA( $q, Y, q_1, G, SCC$ )
7: else //  $A$  is a nonterminal
8:   if  $A \in SCC_i$  and  $|SCC_i| > 1$  then
9:     if  $SCC_i$  is “left” then
10:      foreach  $C \rightarrow X_1 \cdots X_m \in P$  such that  $C \in SCC_i$  and  $X_1 \cdots X_m \notin SCC_i$ 
11:         $q_C = \text{lookup}(C, Q)$ ;
12:        MAKEFA( $q_0, X_1 \cdots X_m, q_C, G, SCC$ )
13:      foreach  $C \rightarrow DX_1 \cdots X_m \in P$  such that  $C, D \in SCC_i$  and  $X_1 \cdots X_m \notin SCC_i$ 
14:         $q_C = \text{lookup}(C, Q)$ ;  $q_D = \text{lookup}(D, Q)$ ;
15:        MAKEFA( $q_D, X_1 \cdots X_m, q_C, G, SCC$ )
16:       $\Delta = \Delta \cup \{(q_A, \varepsilon, q_1)\}$ 
17:    else //  $SCC_i$  is either “right” or “cyclic”
18:      foreach  $C \rightarrow X_1 \cdots X_m \in P$  such that  $C \in SCC_i$  and  $X_1 \cdots X_m \notin SCC_i$ 
19:         $q_C = \text{lookup}(C, Q)$ ;
20:        MAKEFA( $q_C, X_1 \cdots X_m, q_1, G, SCC$ )
21:      foreach  $C \rightarrow X_1 \cdots X_m D \in P$  such that  $C, D \in SCC_i$  and  $X_1 \cdots X_m \notin SCC_i$ 
22:         $q_C = \text{lookup}(C, Q)$ ;  $q_D = \text{lookup}(D, Q)$ ;
23:        MAKEFA( $q_C, X_1 \cdots X_m, q_D, G, SCC$ )
24:       $\Delta = \Delta \cup \{(q_0, \varepsilon, q_A)\}$ 
25:    else
26:      foreach  $A \rightarrow X \in P$  do
27:        MAKEFA( $q_0, X, q_1, G, SCC$ )

```

Figure A.9: Converting a strongly regular grammar into a finite automata)

The resulting strongly regular grammar is $G' = (\{a, b, c\}, \{A, A'\}, A, P')$. Note that $L(G') = a^*cb^*$ while $L(G) = \{a^n cb^n \mid n \geq 0\}$. Therefore, it is easy to see that $L(G) \subseteq L(G')$. The transformation broke the synchronization between the number of a 's and b 's ($\#a$'s $= \#b$'s) by allowing an arbitrary number of a 's and b 's. \square

Next, we show in Figure A.9 the procedure to convert a strongly regular grammar to a finite-state automaton. The algorithm is presented as described in [15]. The main procedure MAKEFA takes five inputs: the initial state, the string, the final state reached on reading the string, the strongly regular grammar, and the mutually recursive nonterminals. We assume a helper function `lookup` that maps nonterminals to automata states. If the nonterminal A is not in the map then a new fresh state q is returned and the pair (A, q) is inserted in the map. Moreover, we add q into Q . Otherwise if there exists already a pair (A, q) then q is returned. Note that since `lookup` can add new states into Q we pass it as an argument.

Starting from the start symbol of the grammar, the procedure MAKEFA descends the grammar (by triggering lines 3-6) until terminals are found (lines 1 and 2), and it creates automata transitions labelled with those terminals. While descending if it encounters a non-recursive nonterminal A it continues recursively for each right-hand side of a production for which A is the left-hand side (lines 26-27). The nontrivial case is when the algorithm encounters a recursive nonterminal symbol A . We describe the case when the set of mutually recursive nonterminals where A belongs to is classified as “left”. The other case when it is either “right” or “cyclic” is symmetric.

Since the set of mutually recursive nonterminals SCC_i is “left” all its productions must be of the form (1) $C \rightarrow \alpha$ or (2) $C \rightarrow D\alpha$, where $C, D \in SCC_i$ and $\alpha \in (\Sigma \cup (V \setminus SCC_i))^+$. The **foreach** loop at

lines 10-12 covers case (1) by descending recursively the right-hand side of the productions whose left-hand side is denoted by C (*i.e.* α) and passing to the recursive call q_C as final state, an automata state assigned to C . The next **foreach** loop at lines 13-15 handles the other case (2) in a similar manner but the first symbol D appearing at the right-hand side is another nonterminal symbol from the same strongly connected component than C . We descend again the right-hand side but providing to the recursive call q_D and q_C (states assigned to D and C) as the initial and final states, respectively.

Appendix B. Intersection of a Context-Free and a Regular Language

The algorithm to check intersection between a CFL and a regular language works on context-free grammars with rules of these forms:

$$A \rightarrow BC \mid a \mid B \mid \varepsilon \quad (\text{B.1})$$

An arbitrary CFG can be converted to a grammar of this form by a linear increase in terms of the size of the original grammar. Let \Rightarrow^* be the reflexive transitive closure of \Rightarrow . For a grammar G in the above form and a set $L \subseteq \Sigma_V^*$ we denote by $pre_G^*(L)$ the set of predecessors of elements in L with respect to the grammar G . That is,

$$pre_G^*(L) = \{\alpha \in \Sigma_V^* \mid \exists \alpha' \in L : \alpha \Rightarrow^* \alpha'\}$$

Starting from a finite-state automaton $A = \langle Q, \Sigma, \Delta, q_0, F \rangle$ that recognises a language L , and a context-free grammar $G = (V, \Sigma, P, S)$, we produce an automaton $A_{pre_G^*(L)}$ recognizing $pre_G^*(L)$ by adding transitions to A according to the following saturation rule:

If $A \rightarrow \beta \in P$ and $\langle q, \beta, q' \rangle \in \Delta$ in the current automaton, add $\langle q, A, q' \rangle$ to Δ .

Given the specific form of our input context-free grammar, we can split the saturation rule into four cases:

1. for each rule $A \rightarrow \varepsilon \in P$ and for each $q_i \in Q$ add $\langle q_i, A, q_i \rangle$ to Δ .
2. for each rule $A \rightarrow a \in P$, if there exists $\langle q_0, a, q_1 \rangle \in \Delta$, then add $\langle q_0, A, q_1 \rangle$ to Δ .
3. for each rule $A \rightarrow B \in P$ ($B \in V$), if there exists $\langle q_0, B, q_1 \rangle \in \Delta$ then add $\langle q_0, A, q_1 \rangle$ to Δ .
4. for each rule $A \rightarrow BC \in P$, if there exists $\langle q_0, B, q_1 \rangle \in \Delta$ and $\langle q_1, C, q_2 \rangle \in \Delta$ then add $\langle q_0, A, q_2 \rangle$ to Δ .

The intersection of a regular language and a context-free grammar can be found by computing first $pre_G^*(L)$ and then checking if there is a transition $\langle q_0, S, q_F \rangle \in \Delta_{A_{pre_G^*(L)}}$. The latter check can be done in constant time so the complexity is the same as the complexity of the pre^* algorithm.

Example 5. Consider the palindrome grammar $G = \langle \{a, b\}, \{A\}, A, P \rangle$ where P is the set of productions: $A \rightarrow aAa \mid bAb \mid a \mid b \mid \varepsilon$, and the finite automaton shown in Figure B.10(a) recognizing the language $L = abba$. We would like to check if $abba \in L(G)$.

First, we generate the automaton that recognizes $pre_G^*(L)$ depicted in Figure B.10(b) and then we check if there is a transition from the initial to the final state labelled with A (that is, $A \rightarrow^* abba$).

In fact, rather than computing $pre_G^*(L)$ directly we normalize our input grammar obtaining $G' = \langle \{a, b\}, \{A, B, C, D, E\}, A, P \rangle$ where P is the set of productions:

$$\begin{array}{lll} A & \rightarrow & CB \mid DE \mid a \mid b \mid \varepsilon \\ B & \rightarrow & AC \end{array} \quad \begin{array}{ll} C & \rightarrow a \\ D & \rightarrow b \end{array} \quad \begin{array}{ll} E & \rightarrow AD \end{array}$$

and compute $pre_{G'}^*(L)$ instead. Note that since there exists a transition from q_0 to q_4 with label A in the automaton shown in Figure B.10(b) we have proven that $abba \in L(G')$.

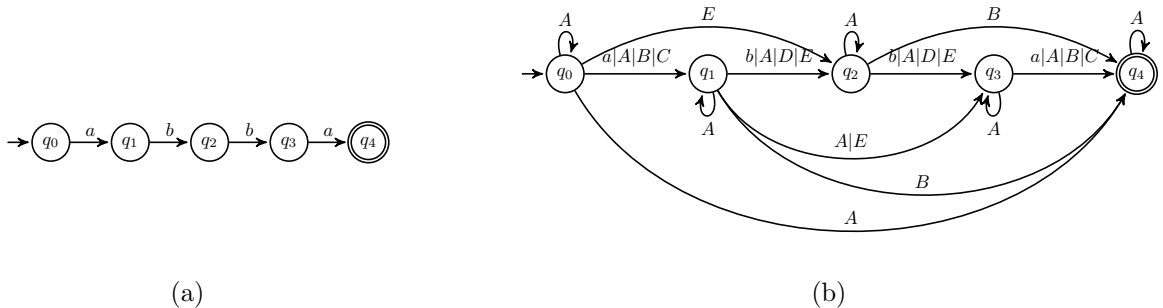


Figure B.10: Two finite automata recognizing $L = abba$ and $pre_{G'}^*(L)$, respectively. The transition $\langle q_0, A, q_4 \rangle$ means that $abba \in L(G')$.

Appendix C. Recursive Multithreaded Programs

Detailed descriptions of the programs used in Section 6’s experimental evaluation, as well as their safety properties, can be found in [13, 4, 17]. This appendix is intended as a self-contained short description.

There are two classes of programs: Erlang programs extracted from textbook algorithms and several variants of a real Bluetooth driver implementation. Table C.2 shows the sizes of the programs after each context-free grammar has been normalized (that is, they satisfy the form in Equation B.1):

- $\#CFGs$: the number of context-free grammars
- $|\Sigma|$ number of terminal symbols
- $|N|$: total number of nonterminal symbols
- $|P|$: total number of grammar productions

Program	$\#CFGs$	$ \Sigma $	$ N $	$ P $	Program	$\#CFGs$	$ \Sigma $	$ N $	$ P $
SharedMem	4	8	138	234	Version 1	7	17	471	804
Mutex	4	22	297	512	Version 2	9	26	1055	1847
RA	2	20	127	205	Version 2 w/ Heuri	9	26	807	1351
Modified RA	5	22	323	530	Version 3 (1A2S)	9	22	746	1292
TNA	3	17	134	204	Version 3 (1A2S) w/ Heuri	8	22	569	938
Banking	3	13	144	244	Version 3 (2A1S)	9	25	1053	1052

Table C.2: Sizes of the programs shown in Table 1(a-b)

Erlang programs. **SharedMem** is the shared memory program shown in detail in Figure 6. **Mutex** is an implementation of the Peterson mutual exclusion protocol where two processes try to acquire a lock. The checked property is that at most one process can be in the critical section at any one time. **RA** is a resource allocator manager that handles “allocate” and “free” requests. We check that the manager cannot allocate more resources to clients than there are currently free resources in the system. **Modified RA** adds some new functionality to the logic of the resource allocator manager. We check the same property used in **RA**. **TNA** is a telephone number analyzer that serves “lookup” and “add number” requests. The property to check is that some programming errors cannot happen. Finally, **Banking** is a toy banking application where users can check a balance as well as deposit and withdraw money. We check that deposits and withdrawals of money are done atomically.

Bluetooth driver [12]. This is a simplified implementation of a Windows NT Bluetooth driver and several variants discussed originally in [17]. The driver keeps track of how many threads are executing in the driver. The driver increments (decrements) atomically a counter whenever a thread enters (exits) the driver. Any thread can try to stop the driver at any time, and after that new threads are not supposed to enter the driver. When the driver checks that no threads are currently executing the driver, a flag is set to true to establish that the driver has been stopped. Other threads must assert this flag is false before they start their work in the driver. There are two dispatch functions that can be executed by the operative system: one that performs I/O in the driver and another to stop the driver. Assuming that threads can asynchronously execute both dispatch functions we check the following race condition: no thread can enter in the driver after the driver has been stopped. **Version 1** and **Version 2** [17] are two buggy versions of the driver implementation. **Version 2 w/ Heuri** is an alternative encoding of **Version 2** introduced by [13] to limit context switches only at basic block boundaries. This makes the verification task easier but it is, in general, unsound as it does not cover all possible behaviours of the driver. **Version 3 (2A1S)** [4] is a safe version after blocking the counterexample found in **Version 2** where two adder and one stopper processes are considered, **Version 3 (1A2S)** is a buggy version with one adder and two stopper processes, and finally, **Version 3 (1A2S) w/ Heuri** is an alternative encoding with the unsound heuristics used in **Version 2 w/ Heuri**.